Multi-cluster parallel job submission: Experiences with Monte Carlo simulations for computational finance on Grid5000

Ian Stokes-Rees, Françoise Baude, Viet Dung Doan, Mireille Bossy

Multi-cluster parallel job submission: Experiences with Monte Carlo simulations for computational finance on Grid5000

Ian STOKES-REES*
Francoise BAUDE
Viet-Dung DOAN
Mireille BOSSY
INRIA Sophia-Antipolis, I3S, Université de Nice Sophia-Antipolis, CNRS
2004 Rte Des Lucioles, BP 93
06902 Sophia Antipolis CEDEX
France
First.Last@inria.fr

September 17, 2007

Abstract

As experience with independent and embarassingly parallel computations in a grid environment mature, it has become possible to explore parallel computing on grids with higher levels of inter-task communication. The PicsouGrid project applies grid computing concepts to computational finance, aiming to leverage heterogeneous resources for both time critical and high volume computations. It utilizes the ProActive Java distributed computing library to parallelize and distribute Monte Carlo option pricing simulations, concurrently utilizing $10^2 - 10^3$ workers. PicsouGrid has been deployed on various grid systems to evaluate its scalability and performance. Issues arising from the heterogeneity and layering of grid infrastructures are addressed via an abstract process model which is applied at each layer. Timings of both the algorithms and the grid infrastructures are carefully measured to provide better insight into the behaviour and utilization of computational grids for this important class of parallel simulation.

Keywords: computational grid, computational finance, grid performance

1 Introduction

The 1973 publication by Black and Scholes[1] of an analytical model for derivative financial products, namely put and call options, ushered in a new era of computational finance. The commoditization, decrease in cost, and increase in computational power of general purpose computing resources has allowed market speculators, financial services firms, economists and mathematicians to develop increasingly advanced models for asset pricing and market behaviour which can be used to identify market opportunities, price products, or estimate risk. The last several years have seen the expansion of hedging tools (as well as portfolio managements tools) for various forms of financial risk (and not just market risk) through the use of derivative financial products. Between large investment banks which require massive computing resources for day trading and over-night risk reporting and the growing market in derivative products which require complex high-dimensional pricing simulations, the need to harness and optimally utilize any and all available computing resources is more pressing than ever. Large computing farms for batch serial execution of financial calculations are now well established within the financial services industry. The PicsouGrid project attempts to introduce grid computing concepts to the domain, to provide automated load balancing, dynamic resource acquisition, fault tolerance, parallelism, and an application framework which can be deployed on heterogeneous underlying resources. Monte Carlo simulations, as used in computational finance, are a particularly interesting category of grid job in that they are generally "dynamically divisible" simulations, consisting of many simple iterative tasks ($> 10^5$) which can be grouped together in arbitrary sizes, while still being part of a larger algorithm with non-trivial issues of latency, inter-task communication, and synchronisation. There are three target audiences for this work: large financial services firms which need to abstract their user and application interface from the underlying resources to provide a uniform, scalable, and robust computing environment; small financial services firms which need the flexibility to utilise diverse and possibly federated heterogeneous computing power; and pricing algorithm developers who want to focus on the pricing algorithm and not

*All correspondence should be sent to Ian Stokes-Rees

worry about resource management issues, or the complexity of coordinating multi-threaded, distributed, parallel programmes.

This paper reports on recent studies into the behaviour of grids for the deployment of large-scale cross-site parallel applications (> 100 cores, $> 5sites$). The ProActive Java parallel/distributed computing library[2] has been used to manage the deployment and synchronization of the parallel pricing algorithms onto the French Grid5000 infrastructure[3], which provides over 3000 cores at 9 sites across the country. The key contributions of this work are empirical studies of intra- and inter-cluster heterogeneity, proposed performance metrics for parallel applications in this domain, and a layered grid process model which clarifies the stages of a grid job and provides a common syntax for timing and logging purposes.

The following section provides some background regarding the ProActive library, the Grid5000 computing environment, the previous versions of PicsouGrid, and the computational structure of Monte Carlo option pricing algorithms. The third section motivates the need for a grid process model and presents a state machine model which can recursively be applied at the various layers found in typical grid environments. The fourth section introduces a number of metrics useful in the domain of parallel grid jobs. The fifth section presents the empirical results of our study, along with observations, analysis, and conclusions. The final section summarises our work and describes the next steps for PicsouGrid and the creation of a performant large scale parallel Monte Carlo simulation environment for the grid.

2 Background

2.1 Project Goals

The PicsouGrid project aims to develop a framework for developing and executing parallel computational finance algorithms. As part of this work, several parallel option pricing algorithms have been developed. Options are derivative financial products which allow the buying and selling of risk related to future price variations. The option buyer has the right (but not obligation) to purchase (for a call option) or sell (for a put option) some asset in the future at a fixed price. Estimates of the option price are based on the well known arbitrage pricing theory: the option price is given by the expected value of the option payoff at its exercise date. For example, the price of a call option is the expected value of the positive part of the difference between the market value of the underlying asset and the asset fixed price at the exercise date. The main challenge in this situation is modelling the future asset price and then estimating the payoff expectation, which is typically done using statistical Monte Carlo simulations and careful selection of the static and dynamic parameters which describe the market and asset. In the context of the work described here, it is sufficient to state that these Monte Carlo simulations consist of 10^5 to 10^7 independent simulations, typically taking anywhere from a few seconds to several minutes for a single asset with a fixed option exercise date (called a *European option*). These *vanilla options* can be computed in parallel, typically in blocks of 10^3 to 10^4 iterations, and then the statistics gathered to estimate option prices. Two variations on vanilla options which introduce much more computational complexity are *basket options*, where a set of underlying assets are considered together, and *American options*, where the option can be exercised at any point up to the contract expiry date. The complexity of a five asset American option compared to an otherwise equivalent single asset European option would be several orders of magnitude greater, taking hours to compute serially. In a liquid market, such delays are generally unacceptable as the market situation is continually changing. Pricing decisions generally need to be made in seconds or at most minutes to be useful to market traders, hence the need to explore alternative strategies for pricing of complex options. Here we consider parallel pricing algorithms, with an effort to address large scale parallelisation with the goal of reducing computation time of a given complex pricing request from hours to minutes.

Innovations over the last decade have made derivative products such as options a key part of global financial markets, partially due to computing advances which have allowed market investors to more accurately price these products. Better option pricing (that being more accurate, more advanced models, and faster results) and hedging provide a market advantage and are therefore of great interest to market traders. To date, there has been limited public discussion on the parallelisation of pricing algorithms, and even less on the computation of option prices in a grid environment. The reason for this is partially due to the trade secrets financial services firms have invested in developing their own pricing models and computing environments to give them advantages over their competitors. This work makes contributions to the domain by implementing and making publicly available various serial and parallel option pricing algorithms, a framework for developing further algorithms, and a flexible grid computing environment to perform calculations.

2.2 ProActive Java Library

PicsouGrid has been developed in Java with the ProActive[2] parallel and distributed computing library. The use of Java allows PicsouGrid to be used in a wide range of computing environments, from standard Windows desktop systems, to large Linux clusters. ProActive implements the *Active Object* model to enable concurrent, asynchronous object access and guarantees of deterministic behaviour. Incorporating ProActive into PicsouGrid requires minimal modification of the frame-

work or specific algorithm implementations. ProActive requires a few constraints on the construction of Objects which will be accessed concurrently, such as empty argument constructors, limited use of self reference in method call parameters, no checked exceptions, and non-primitive, non-final return values for methods. In return, ProActive provides a generic object factory which will dynamically instantiate a "reified" version of any desired object on any available host, while providing the application with a stub which can be utilised exactly as an instance of the standard object. The reified object consists of the proxy stub, a wrapper object, a message queue, and a wrapped instance of the actual object. Only the proxy stub is on the local node. The wrapper object is started by ProActive on the remote node (either specified explicitly as a parameter to the object factory, or selected automatically by ProActive), and contains a message queue for all public method calls on the object, and finally the wrapped object itself. PicsouGrid makes heavy use of the Typed Group Communications features of ProActive[4] to enable parallel execution of Monte Carlo simulations, as well as broadcast and gather-cast features for uniform configuration and interrogation of worker object states. Through the use of ProActive it is possible to run simulations on a single machine, a desktop grid, a traditional cluster, or on a full grid environment without any additional configuration effort in the application. The ProActive deployment mechanism automatically contacts and initiates services and objects on the remote nodes[5].

2.3 PicsouGrid Architecture

Previous versions of PicsouGrid have focused on fault tolerance and the use of JavaSpaces to provide a shared data object environment as the primary strategy for coordinating parallel Monte Carlo simulations across a set of workers[6]. A master-worker architectural approach was followed with three layers: master, sub-masters, and workers. The user accessed the system through the master, which in turn could be used to initiate sub-masters associated with specific groups of computing resources (typically clusters), each of which managed their own local workers. When workers failed, this would be detected via a "heartbeat" mechanism and the worker replaced from a reserve pool. When sub-masters failed they too would be replaced by the master. This architecture was developed for vanilla option pricing algorithms where each worker task consisted of a given number of iterations of the specified algorithm with a fixed set of parameters. When a worker or sub-server was lost, it was simply a matter of re-allocating the "lost" iterations to an existing or new replacement worker or sub-server to continue. "State" for any set of workers (or sub-masters) consisted exclusively of the number of iterations they had been allocated and not yet completed. With the addition of more complex algorithms for American option pricing, it is necessary to do computations in stages, and sometimes short iterative cycles where all workers must be updated with newly calculated values from the previous iteration. In this situation it is more difficult to synchronize and recover from worker failures at arbitrary points in the overall computation. The latest version of PicsouGrid shifts the focus from fault tolerance of workers and the overall infrastructure to autonomy, scalability, and efficient distribution of tasks for complex option pricing algorithms. This is achieved via a mechanism where *masters* and *workers* are merged into general *simulators*. A simulator has the ability either to complete any portion of the algorithmic computation itself, or similarly route it to a worker set. This model provides a greater degree of flexibility in that computations can be initiated on a given object without regard as to whether that object will complete the computation itself or draw on a local, and possibly externally inaccessible, pool of "sub"-simulators. Furthermore, any computation by a simulator is *voluntary*, without expectation of completion. This means a given simulator may *request* that its set of sub-simulators complete some stage of a computation, however it is then up to those sub-simulators to decide to do the work and return the results. The lead simulator does not expect the results for work which has been allocated, which allows for the possibility that a given sub-simulator may fail or execute the work packet very slowly. Instead, the lead simulator continues to hand out work packets until the job is complete. The asynchronous, non-blocking nature of active objects allows the fastest simulators to take more work packets and for merging of results to occur as those packets are completed, without any need for global synchronization.

2.4 Grid5000

Grid5000 is a research-oriented grid based at 9 centres in France, and consisting of 16 cluster and over 3000 cores[3]. This research grid has a focus on parallelism and as such high performance networking is available within clusters, typically consisting of Myrinet or Infiniband interconnects. Furthermore, sites are interconnected with 10Gb/s networking, allowing for inter-site parallel computing. As a research environment, Grid5000 is on a private network and not accessible by the Internet, except through per-job port forwarding. The focus on parallelism means a specialised scheduler has been developed which can reserve blocks of nodes either on a "best effort" basis (as many nodes as possible up to a maximum), or a fixed number of nodes at a specific time and for a specific duration. In Grid5000 parlance a node is a physical host, with however many cores that host happens to have being allocated to the user job. All scheduling and reservations are done at the cluster level, with some "helper" interfaces which will concurrently launch reservations on a group of clusters, but make no effort to coordinate reservations – this is left to the user

to manage manually. Finally, Grid5000 provides only a basic interface for launching parallel jobs via a user specified script which executes on a "leader" node for each cluster. The MPI-style NODEFILE variable points to a file containing the list of all nodes (by private network hostname) allocated by the cluster for the job. It is then up to this user script executing on each leader node to utilise this list of nodes to initiate the parallel computation. There is no inherent facility for cross-cluster coordination, no global file space (typically all files are synchronized to cluster- or site-local network file systems), and no automated logging facilities beyond capturing the "standard output" and "standard error" for the script executing on the leader node. While this work focuses on the infrastructure provided by Grid5000, since this was used for the results presented in Section 5, many of these characteristics are also shared with LCG[7] and OSG[8], perhaps with the key exception that they both provide grid-level scheduling. Grid scheduling allows jobs to be submitted at the grid level, and then allocated to an appropriate site and cluster based on the characteristics described in the job meta-data. While Grid5000 lacks this feature, LCG and OSG's grid level scheduling does not provide any facility for coordinated cross-site reservations for distributed parallel grid computing. One special feature Grid5000 provides, and which has not been utilised in the work here, is the ability to create a custom system image which will be used by the job. This image can then be replicated to all clusters and, to a degree, a homogeneous grid operating environment can be created for the user's grid job across all reserved nodes, with Grid5000 managing the reboot and selection of system image in preparation for the job.

2.5 Parallel Monte Carlo Simulation

Monte Carlo simulations rely on a large number of independent simulations of a model which makes use of randomly generated numbers. Aggregate behaviour of the model is then determined by averaging the results, taking into consideration the variance of the independent simulations. In a parallel environment this suggests that the simulations are conducted concurrently by the available parallel processors. In a traditional homogeneous parallel environment there is an implicit assumption that these simulations all take the same amount of time, so the division of the simulations is uniform, and typically I total iterations are divided by P available processors at each stage, and each processor handles I/P iterations. Complex models proceed in stages, possibly with convergence iterations, where the results from one stage must be gathered from all processors, merged, and some updated parameters calculated and distributed back to each processor. This communication overhead can be a major bottleneck for parallel algorithm implementations, and the impact increases with the number of parallel processors.

In a grid environment it is not possible to assume the processors are uniform. In fact, during computation some processors may fail or proceed very slowly, in which case the remaining processors should be able to adjust their workload appropriately. This suggests applying a factor F to the number of processors P, such that I iterations are divided by $F \times P$. In a uniform environment, each processor would handle F packets of size $I/(F \times P)$, however in a heterogeneous environment the faster processors would acquire $> F$ packets, and the slower processors $< F$, thus providing a degree of load balancing. The selection of packet size has an impact on the waiting time for the last packet to be returned (where smaller packets are better) *versus* the communications overhead of many packets (where larger packets are better). Tuning this depends on many factors and is not an objective of the work presented here, but will be analysed in the future. Related to this issue of optimal packet size is the degradation in speed-up with additional processors. This is the classic problem of parallel computing, again related to the CCR: Communications-to-Computation Ratio. The goal of the work presented here is to lay a foundation for a maximum number of coordinated parallel processors to be utilised in a grid environment, without regard to the ultimate efficiency of the computation in this configuration. See Section 4 for more discussion on the experimental design used for this work and the definition we adopt for efficiency.

Some work which has been done on parallel grid deployments of computational finance algorithms have used the Longstaff-Schwartz[9], Picazo[10], and Ibanez-Zapatero[11] algorithms. In particular, [12] looks at using Longstaff-Schwartz in a grid environment, however with only a maximum of 12 processors. [13] uses MPI with no more than 8 tightly coupled identical CPUs for the Boyle quasi-Monte Carlo stochastic mesh algorithm[14]. [15] uses an SGI 32-CPU SMP machine for the original Broadie-Glasserman stochastic mesh algorithm[16]. These efforts have shown the viability of parallel algorithms, with efficiencies ranging from $70 - 95\%$ (depending on the number of parallel CPUs). While all of these cite market time constraints as a key issue for the parallelization of Monte Carlo-based American option pricing algorithms, none address the issues of deployment and computation on heterogeneous, distributed, dynamic, and large scale grid infrastructures, aspects which are key goals of the PicsouGrid project.

3 Grid Process Model

The Unix process model[17], with its three primary state of READY, RUNNING, and BLOCKED, provides a common basis for the implementation of POSIX operating system kernels, and an understanding of the behaviour of a process in a pre-emptive multi-tasking operating system. Users and developers have a clear understanding of the meaning of system time (time spent on kernel method

calls), user time (time spent executing user code), and wait/block time (time spent blocking for other processes or system input/output). There are analogous requirements in a grid environment where users, developers, and system administrators need to understand what state and stage a "grid job" is in at any given time, and from a post-mortem perspective be able to analyse the full job life-cycle. The federated nature and automated late-allocation of disperse and heterogeneous computing resources to the execution of a grid job make it difficult to achieve this. While some effort has been made to produce a common semantic model for life-cycles and process interaction on the grid (*e.g.* SAGA[18] and CDDLM[19]), these have primarily focused on APIs or new SOA models for service interaction of deployed applications, rather than a more modest goal of a logical semantic model for grid jobs, when viewed as an extension of the traditional operating system process model.

Within the context of our work on a parallel grid Monte Carlo simulation environment, it became clear that a common model was necessary in order to understand latencies and behaviour within the various layers of the grid environment on which our simulations were being run. The remainder of this section defines terms related to our grid process model and presents a recursive state machine which has been used for tracking the life cycle of grid jobs. In fact, the model we have developed here is not specific to parallel Monte Carlo simulations on the grid, but is generally applicable for describing the state of a "grid process" where many sites, clusters, hosts, and "operating system processes" may concurrently be involved.

3.1 Grid Tasks and Jobs

There is no commonly agreed model for a task in a grid environment. As a result, it is difficult to discuss and design systems which manage the life-cycle of a programme executing on a grid. This is partially due to the lack of a common definition of a "grid task", and its scope. The GGF Grid Scheduling Dictionary[20] offers two short definitions which provide a starting point:

> **Job** An application or task performed on High Performance Computing resources. A Job may be composed of steps/sections as individual schedulable entities.
>
> **Task** A specific piece of work required to be done as part of a job or application.

Besides the ambiguity introduced by suggesting that a *job* is also a *task*, these definitions do not provide sufficient semantic clarity for distinguishing between work-flows and parallel executions. We therefore feel it is necessary to augment these terms to contain the concept of co-scheduling of resources to provide coordinated parallel access, possibly across geographically disperse resources. We propose the following definitions:

> **Basic Task** A specific piece of work required to be done as part of a job, providing the most basic computational unit of the job, and designed for serial execution on a single hardware processor. The scheduling and management of a *basic task* may be handled by the grid infrastructure, or delegated to a *grid job*.
>
> **Grid Job** A task or a set of tasks to be completed by a grid infrastructure, containing meta-data to facilitate the management of the task both by the user and a specific grid infrastructure.
>
> **Work-flow Job** A *grid job* containing a set of dependencies on its constituent basic tasks or other *grid jobs* and which is responsible for the coordinated execution and input/output management of those sub-jobs or sub-tasks.
>
> **Parallel Job** A *grid job* which requires the coordinated parallel execution and communication of a set of *basic tasks*.

With this set of definitions we consider a simple parallel Monte Carlo simulation to consist of a *parallel job* with a set of coordinated *basic tasks*, such that the grid infrastructure provides a set of concurrent computing resources on which the simulation can initiate the parallel computation. A more complex phased Monte Carlo simulation would consist of a *work-flow job* where each phase of the work-flow would consist of a *parallel job*, executing that phase's parallel computation as part of the larger Monte Carlo simulation. The grid infrastructure is then responsible for the appropriate selection, reservation, and allocation/binding of the grid computing resources to the simulation job (whether simple or complex), based on the requirements described within the job itself.

3.2 Recursive Layered State Machine

Figure 1 indicates the system layers typically found in a grid environment and through which a grid job will execute. For a basic grid job, this will consist of one subprocess at each layer. It is possible that the *Site* layer will not always be present, with *Clusters* being accessed directly by the grid infrastructure. The visibility of a particular *Core*, in contrast to the *Host* in which it exists, also may not be distinguishable. Some clusters may allocate resources on a "per-host" basis, with all cores available for the executing task, while others may allocate a number of tasks to a particular host up to the number of physical cores available, trusting the host operating system to correctly schedule each grid task (executing as an independent operating-system-level process) to a different core. Finally, the concept of a *VM* (virtual machine), whether a user-level VM such as Java or an operating system level VM such as Xen or VMWare, either may

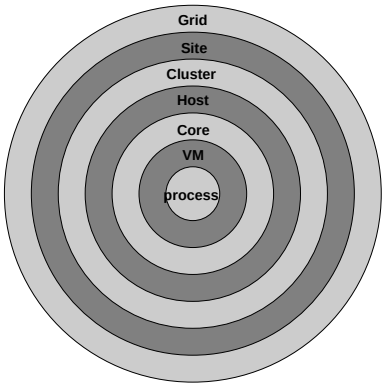

Figure 1: *Example of the various layers through which a grid job executes.*

not exist within the grid environment, or may replace the concept of a core, with one VM allocated to each core within the host, and the host (or cluster) then scheduling grid tasks on a "one-per-VM" basis.

It should be noted that Figure 1 only illustrates system level layers, predominantly representing the layers of physical hardware and networking. There are also the various layers of software, such as the grid framework, the local cluster management system, the operating system, and any application framework which may simultaneously be in use.

As a brief anecdotal example of this layering situation for a parallel grid job, consider the single command shown in Listing 1. This initiates the parallel execution of a Monte Carlo simulation for a simple European Option using the job launcher `mygridsub` which attempts to submit the specified job to all Grid5000 clusters at the same time. The parameter `nodes=30` requests up to 30 nodes (equivalent to hosts) from each cluster, on a best effort basis, while `-r "2007-02-05 6:05:00"` is the timing request for the job to start 5 February 2007 at 6:05am CET, and to run for 1 hour (`walltime=1`). The parameter `es-bench1e9` is a script file which performs the actual simulation.

Once the Grid5000 infrastructure fulfills its reservation request, the `es-bench1e9` script is initiated on *one* of the worker nodes on each cluster which has agreed to provide resources. Listing 2 shows the command which is actually executed on this leader node. From this it can be seen the grid infrastructure has added another layer of wrapping with the `oarexecuser.sh` script, and the PicsouGrid application framework has added `script-wrapper`. The `script-wrapper`, in turn, is responsible for accessing all the subordinate worker nodes allocated within the particular cluster (the list of these are known via a file whose location is given by `OAR_NODEFILE`, which is used by a sub-script `spread-task`), and then spawning one worker simulator per physical processor core available on the particular node (host) (via yet another sub-script `all-cpus`).

Only at this point (and coordinating when this point has occurred is non-trivial) can the primary parallel simulation commence, now with each of the distributed worker nodes properly initialised. In total, eight levels of nested software scripts and six levels of system infrastructure have been traversed in order to get from a single grid submit node to the thousands of distributed grid cores where the parallel compute job is finally executed. When working in a cluster or grid environment many of the aspects which can be easily assumed in a single node environment must be made explicit, and the staging of execution is managed with possible time delays for synchronisation and queuing, and on completion it is necessary to properly "tidy-up" to return the collective results. Taking these various factors into consideration, a five stage model is proposed which is applied at each layer of the grid infrastructure. The stages, in order, are defined as follows:

CREATE prepares a definition of the process to be executed at this layer.

BIND associates the definition, possibly based on constraints within the definition, with a particular system at this layer.

PREPARE stages and data required for execution to the local system which the definition has been bound to and does any pre-execution configuration of the system.

EXECUTE runs the programme found in the definition. This may require recursing to lower layers (and subsequently starting from that layer's CREATE stage). In a parallel context, it is at this stage where the transition from serial to parallel execution takes place, possibly multiple times for the completion of this stage's execution.

CLEAR cleans the system and post-processes any results for their return to the caller (*e.g.* next higher layer).

The grid infrastructure handles the transition from one stage to the next. To accommodate the pipelined and possibly suspended life-cycle of a grid job it is not possible to consider each stage as being an atomic operation. Rather, it is more practical to add entry and exit states to each stage. In this manner, three states are possible for each stage: READY, which is the entry state; ACTIVE, which represents the stage being actively processed; and DONE, when the stage has been completed and transition to the next stage is possible. A grid job starts in the CREATE.READY state, which can be seen as a "blank" grid

```
mygridsub -r "2007-02-05␣6:05:00" -l walltime=1,nodes=30 es-bench1e9
```

Listing 1: *Executing parallel Monte Carlo simulation on Grid5000*

```
/bin/sh -c /usr/lib/oar/oarexecuser.sh /tmp/OAR_59658 30 59658 istokes-rees \
/bin/bash ~/proc/fgrillon1.nancy.grid5000.fr/submit N script-wrapper \
~/bin/script-wrapper fgrillon1.nancy.grid5000.fr \
~/es-bench1e9
```

Listing 2: *Command executed on leader node for each Grid5000 cluster*

job. Once the system or user has completed their definition of the actions for that layer (done by entering CREATE.ACTIVE), the grid job finishes in the CREATE.DONE state. At some later point, the grid infrastructure is able to bind the job to a resource, and later still prepare the bound node(s) for execution. When the node(s) are ready the grid job can execute, and finally, once the execution is complete, the grid job can be cleared.

We have developed this model to be applied recursively at the various layers shown in Figure 1. The layering also includes the software systems, and is arbitrary to a particular grid environment. For example, a grid job could be in the state "CLUSTER/BIND.READY", indicating that a cluster-level job description has been prepared, and now the grid job (or this portion of it) is waiting for the cluster layer of the grid infrastructure to make a binding decision to submit the job to a particular queue. The queue, in turn, would have to allocate the job to a particular host, and so on. In the case of distributed asynchronous parallelism, where different parts of the computation may be in different states, layers can be enumerated such as "CORE[1]/EXECUTE.READY" and "CORE[2]/EXECUTE.DONE". The context for a particular state when there are parallel processes as part of the same grid job is based on the enumerated state of the higher level layers, noting that recursion to lower layers only happens when a particular layer is in the EXECUTE.ACTIVE state. In this way, a notation using the prepended chain of higher layers and commencing with the grid job identifier is possible. In a URL context, some examples of this are shown in Listing 3.

This example shows generic layer labels, as taken from Figure 1, however these can also be replaced by a specific label. For example, the state label `http://www.example.com/grid/jobs/Job2030/sophia.inria.fr/helios/helios22/cpu0/jvm-15542/PREPARE.ACTIVE` would tell us that part of grid job 2030 is executing at the `sophia.inria.fr` site, on the `helios` cluster, on the `helios22` node, running on `cpu0`, and the Java JVM with process ID 15542. Of course if job 2030 is a parallel job, other parts of this job may be in other states on other sites and hosts.

While this model has been developed with the intention to incorporate it into a larger grid workload management system, the current model is only used for logging, time stamping, a monitoring. In many cases (*e.g.* Grid5000 and EGEE) we do not have access to the internals of the grid infrastructure and either do not have visibility of some of the state transitions or are only able to identify state transitions and time stamps during post-processing of grid job logs made available once the job is complete.

This model has allowed us to gather behaviour and performance details for a consistent comparison between three key aspects of any parallel grid application: the grid infrastructure impact; the parallelization framework; and the core application code. It is the basis for all the monitoring and timing information which is provided in the results presented in Section 5. We finish this overview with Figure 2 which is a simplified grid network snapshot showing the state of various entities contributing to a fictitious small parallel computation. It shows two sites, each with two clusters. Three of the clusters have started executing the grid job on their worker nodes, and those six workers are in different states, while one cluster is making binding decisions regarding which workers to execute on.

4 Grid Efficiency Metrics

This work has focused on establishing a foundation for coupled parallel simulation on the grid, and as such does not focus on parallel speed-up *per se*. To simplify the experiments presented here and to highlight the issues introduced by the grid infrastructure and parallel computing environment we have only executed the first stage of the parallel computation, and removed the synchronization at the end of the Monte Carlo simulations. In this way, the experimental jobs appear to be embarrassingly parallel. The work presented here focuses on the capabilities and characteristics of the underlying grid infrastructure to provide for such application-level parallelism. In this environment, we define four metrics relevant to our problem domain:

Time window unit-job through put This metric counts the number of "unit jobs" executed by the grid infrastructure in a fixed time window. Typically the time window is taken from the start of the earliest computation to the end of the last computation, although this can be "time shifted" to align each cluster start-time as $t = 0$.

```
http://www.example.com/grid/jobs/Job1000/SITE/BIND.ACTIVE
http://www.example.com/grid/jobs/Job2030/CLUSTER/HOST/JVM[1]/PREPARE.ACTIVE
http://www.example.com/grid/jobs/Job2030/CLUSTER/HOST/JVM[2]/EXECUTE.DONE
http://www.example.com/grid/jobs/Job2030/CLUSTER/HOST/JVM[3]/EXECUTE.ACTIVE
http://www.example.com/grid/jobs/Job2750/SITE/HOST/CORE[1]/EXECUTE.ACTIVE
```

Listing 3: *Examples of unified grid job states as URLs*

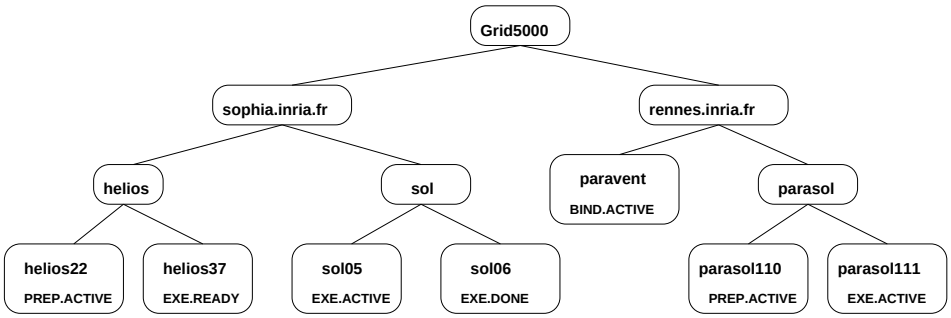

Figure 2: *A simple example of a parallel grid job state snapshot on Grid5000. Nodes without an explicit state are implicitly in the state EXE.ACTIVE, which is the only valid state for entering into a lower layer.*

A "unit job" is some standard job unit which is representative of the application and clearly defined. This would always take the same amount of time for serial execution. This metric measures the capacity and bulk efficiency of the grid infrastructure for a particular grid task. If the number of grid nodes (processors/cores) is somehow fixed, this gives a comparative performance measure of the grid.

Speed-up efficiency limit With some reference system serial calculation time for a unit job, the speed-up efficiency is defined as the time taken for the reference system to process N unit-jobs divided by the total occupancy time at a particular grid layer required to compute the same N unit-jobs. The metric assumes zero communication time for parallel jobs. For example, a cluster containing 40 cores is occupied for 70 seconds, and in this time completes 200 unit jobs. The reference system serial calculation time for a unit job is 1.2 seconds. The speed-up efficiency limit for the given cluster is $(200 \times 1.2)/(40 \times 70) = 240/280 = 85.7\%$. Of course it is possible that the unit-job computation time for a given system is *better* than that of the reference system, in which case the speed-up efficiency limit may be $> 100\%$. This metric is in contrast to the common definition of speed-up which simply calculates the parallel compute time for a homogeneous cluster of N systems compared to the serial compute time on a single processor from the same system. In this case the speed-up limit would always be N and the speed-up efficiency limit 1. Equation 1 defines this metric, where $n_{unitJob}$ is the total number of unit jobs completed by the grid system, n_{procs} is the number of processors contributing to the total computation time, and t_i represents the wall time of the occupancy of that layer of the grid. For instance, efficiency at the level of each core would be calcualted by summing the core occupancy times for each processor, while the host efficiency would be the total time the particular host for that processor was in use during the calculation of the unit job (even if the particular processor was finished), and the cluster-level efficiency would take the occupancy time of the full cluster for each unit-job, regardless of whether the host or processor occupancy time was much shorter.

$$\frac{n_{unitJob} \times t_{ref}}{\sum_{i=1}^{n_{procs}} t_i} \tag{1}$$

Weighted speed-up efficiency limit When there is some knowledge of the relative performance of each processor within the grid, the weighted speed-up efficiency can be calculated, which takes into account the best performance which can be expected from each contributing processor. Equation 2 defines this, where w_i is the weighting factor and is proportional to the relative performance of the processor (higher values mean higher performance). We do not make use of this metric here as no common benchmarks were available at the time the experiments were run. It is planned to build a profile of every node and utilise this in performance prediction and *a priori* load partitioning.

$$\frac{n_{unitJob} \times t_{ref}}{\sum_{i=1}^{n_{procs}} t_i / w_i} \tag{2}$$

Occupancy efficiency This measures what fraction of the total time available for computation was actually used by the application, measured at the various layers within the grid. This is defined by Equation 3, where $n_{compUnits}$ indicate the number of computational units (*e.g.* hosts, cores, VMs, threads, processes) available at

that layer. The figures found in Section 5 show $node_{eff}$ and sim_{eff} which are the node and simulator occupancy efficiency respectively. Values less than 100% indicate that wastage has occurred and the application had a reservation for a computational unit which it did not utilise.

$$\frac{\sum_{i=1}^{n_{compUnits}} t_i}{n_{compUnits} \times t_{reservation}} \quad (3)$$

5 Parallel Experiments on the Grid

The starting point for discussing parallel Monte Carlo simulation on the grid is to understand an ideal situation. Ideally all available computing resources would be used at 100% of capacity for the duration of their reservation performing exclusively Monte Carlo simulations. The time to merge results would be zero, and there would be no communications overhead, latencies, or blocking due to synchronization or bottle necks. This is the classic definition of "linear speed-up", modified for a grid environment by adjusting expected "optimal" results according to the relative power of each individual resource (*i.e.* weighting results according to some benchmark). In reality, as discussed in Section 4, there are many parameters which have an impact on the actual performance of a parallel Monte Carlo simulation. As the following results will show, predictable coordinated access to resources within a single cluster can be difficult, and synchronization of resources between clusters or sites even harder. Due to this observation, the work here focused on identifying the issues which lead to poor resource synchronization, and to facilitate evaluation of resource capability. In order to do this, the following results eliminate coordinated simulation and merging of results, and only initiate independent partial Monte Carlo simulations. All the following experiments were performed in early March 2007 on Grid5000, using all available sites, clusters, and nodes. The figure headings show the statistics for the time spent in the states NODE.EXECUTE.ACTIVE and SIMULATOR.EXECUTE.ACTIVE in the form $node = (M, S)s$ and $sim = (M, S)s$ where M is the mean time in seconds and S is the standard deviation. $node_{eff}$ is the node occupancy efficiency, and sim_{eff} the simulator occupancy efficiency, as defined in Equation 3. The "simulator" is the part of the application where the Monte Carlo simulation is executed, excluding any software startup (*i.e.* JVM initiation), configuration, and tear-down time.

Figure 3 shows an almost ideal situation where 60 cores running on 30 hosts from the same cluster all start processing within a second of each other, run their allocated simulations for the same duration (around 90 seconds, shown by the black "life line"), and complete in a time window of a few seconds. This provides a simulation efficiency of 92.7%, and we take this to be our "cluster-internal" efficiency standard.

By contrast, some clusters showed node (host) and

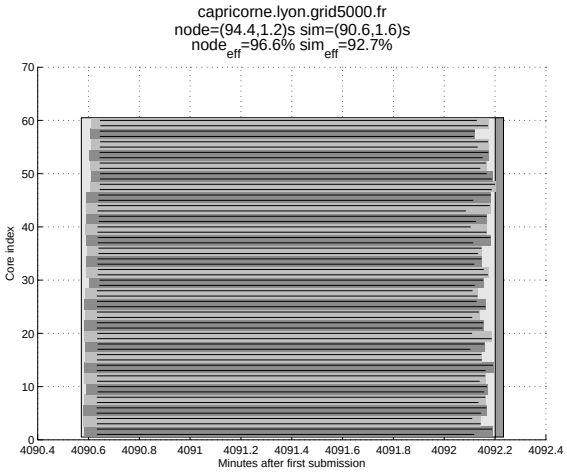

Figure 3: *Realistic optimal parallel Monte Carlo simulation, with 92.7% occupancy efficiency.*

core start and finish windows of several minutes, as seen in Figure 4. This particular example consists of 240 dual-CPU nodes, representing 480 cores. A simulation efficiency of only 43.1% was achieved, indicating that the resources were idle for the majority of the reservation time (the time outside of the black life-lines). Furthermore, the majority of this idle time was in the finishing window, where only a few inexplicably slow nodes delayed completion of the computation on the node block within the cluster. The space prior to the start of the simulator life-line is due to grid infrastructure delays in launching the node- or core-level process (*i.e.* due to delays in the CLUSTER.PREPARE stage) – again, wasted computing time when the job held a reservation for the node and core, but failed to utilise it.

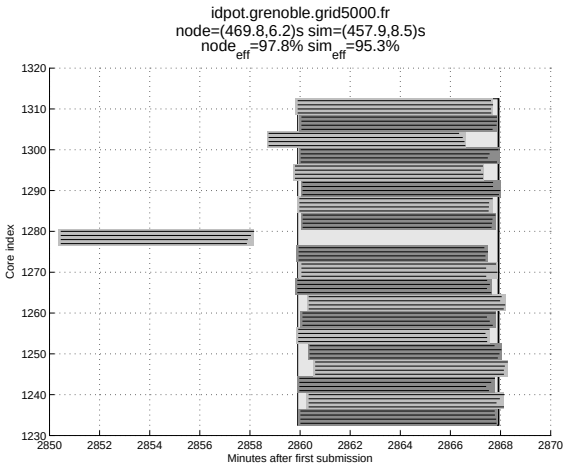

Figure 5: *NTP configuration problems leading to clock skew of seconds to several minutes on certain nodes*

Another common issue across many sites within

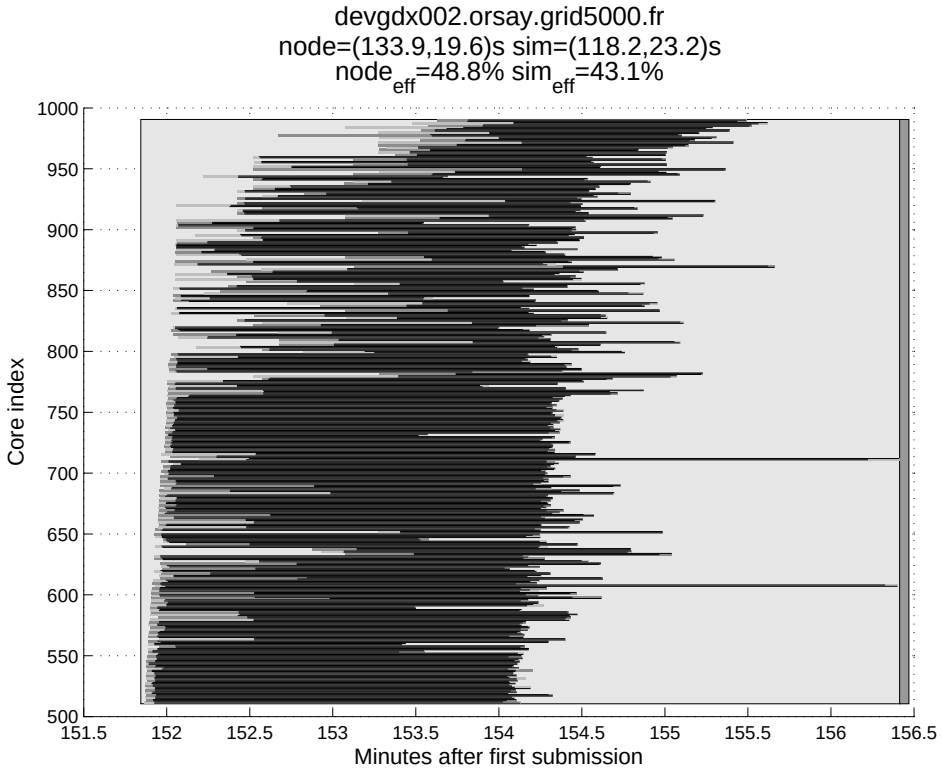

Figure 4: *Cluster exhibiting significant variation in node start and finish times, resulting in an occupancy efficiency of* $< 50\%$.

Grid5000 was the clock-skew due to poorly configured NTP services, or miss configured time-zones (*i.e.* certain nodes believing they are in a different time zone and therefore reporting a time-stamp exactly one hour different from their cluster neighbours). This can be seen in Figure 5. Many parallel and distributed computing libraries rely heavily on well-synchronized timestamps, not to mention network services such as NFS. The effect of this clock-skew is unpredictable but certainly undesirable, not to mention the difficulty it introduces when attempting to take timestamps and track performance, as these activities typically must be done locally due to the complexity of trying to centralise all logging within a large distributed parallel computing system.

Figure 6 shows a situation where a supposedly uniform cluster of dual-core dual-CPU 2.2 GHz AMD Opteron nodes had a dramatic variance in the completion time, both on the inter-node (between hosts), and intra-node (different cores within the same host) completion time. This led to an average computation completion time of 99 seconds, with a standard deviation of 25 seconds, and resulted in only a 64.6% computation efficiency. Approximately two thirds of the nodes completed their simulations in 82 seconds, however the remaining third took around 133 seconds. If this sort of behaviour was very rare it could be ignored, however this study suggests that it occurs with sufficient frequency to be a major concern for communicating parallel computations, and even for embarrassingly parallel computations where the division of the workload may be done based on inaccurate static measures for the relative performance of a node.

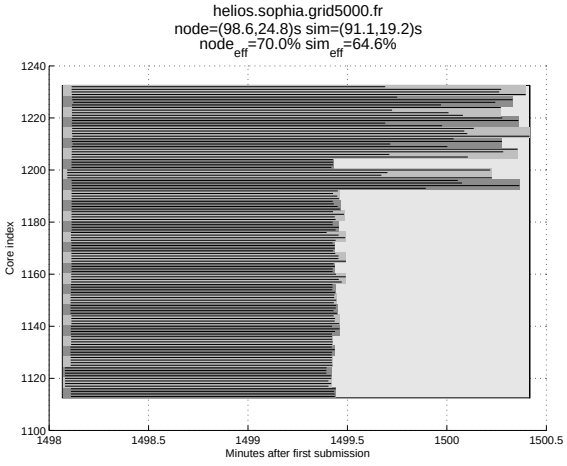

Figure 6: *Dedicated cluster of uniform nodes showing wide variation in performance*

Finally, we return to the question of coordinated cross-cluster (and cross-site) parallel computing. Besides inherent technical issues present when attempting regular communications across long network segments, it is dif-

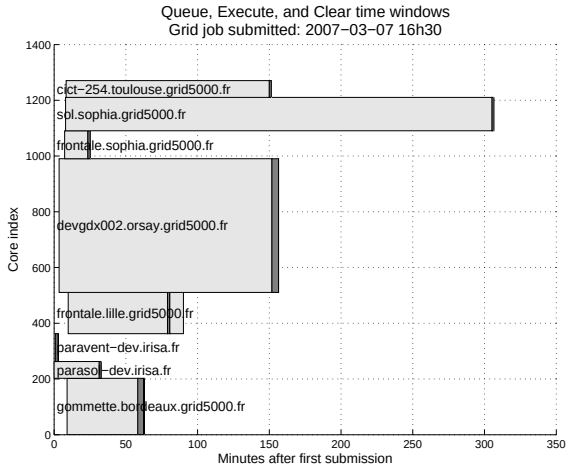

Figure 7: *Light grey boxes indicate queuing or clearing time, dark grey boxes indicate execution time. This graph illustrates the difficulty in coordinated cross-site on-demand parallel computing.*

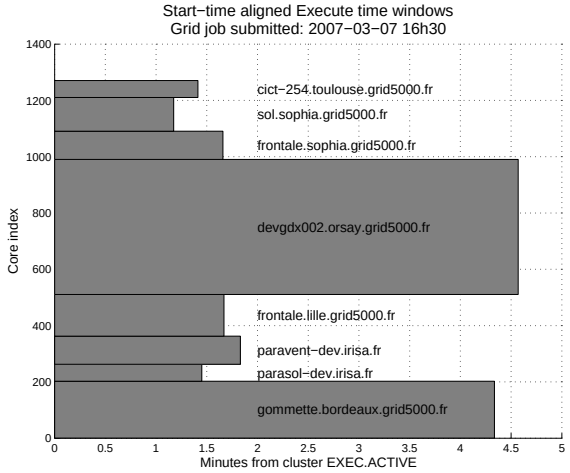

Figure 8: *Cluster start-time aligned execution phase, showing overall (time-shifted) "grid" time window for grid job execution. This shows wasted computing power as fast nodes sit idle waiting for slow nodes to complete.*

ficult to satisfy on demand requests for grid-wide multi-node reservations. There is the initial challenge of immediate availability, and then the subsequent challenge of promptly completing and confirming the distributed node reservations and the requisite site, node, or cluster preparation (pre-execution configuration). Figure 7 shows an example of such a situation for a 1270-core multi-cluster job, where a request for approximately 80% of the reported available nodes from each cluster was made at a fixed point in time, and most clusters took over an hour before this request could be fulfilled. One cluster took two and a half days (not shown in figure due to effect on time scale). This is not a surprising result given the nature of Grid5000 and the style of computations which are done on it, namely experimental (therefore usually less than one hour) parallel and distributed computing. At any given time it is expected that the clusters will be full with jobs which require blocks of nodes, and for the queue to contain multi-node jobs as well, therefore newly submitted jobs would expect to wait at least several hours to both make it to the front of the queue and for the requisite block of nodes to be available. While Grid5000 provides simultaneous submission of jobs, it does not coordinate reservations and queues, so the individual clusters became available at different times. Using normalised cluster start-times, all the unit-job computations took place in a 274.2 second time window, for a total execute stage time block of 1270 cores $\times$ 274.2 seconds = 348234 core·seconds = 96.7 core·hours. Compared with a reference unit-job execution time of 67.3 seconds, the speed-up efficiency limit, as given in Equation 1, is (1270 cores $\times$ 67.3 seconds)/ 96.7 core·hours = 25.5%. Figure 8 shows the start-time aligned execution phase for each cluster. An obvious conclusion from this is the need to partition the computational load in such a way that the faster nodes are given more work, rather than remain idle while the grid job waits for the slowest nodes to complete.

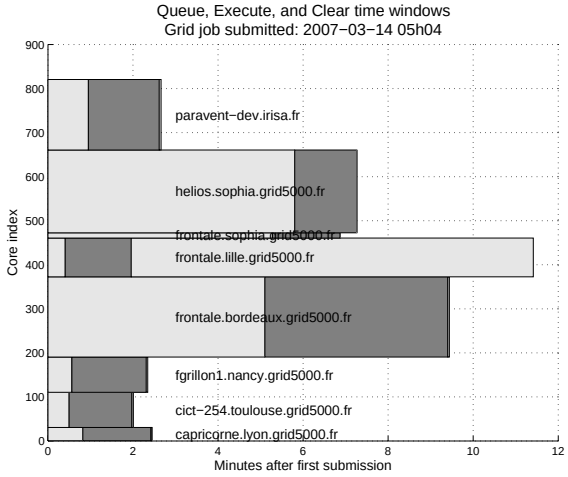

Figure 9: *Even with reservations for cross-site synchronized parallel computing it is difficult to coordinate resource acquisition. Light grey boxes show waiting time, dark boxes show execution time.*

A more realistic multi-cluster parallel grid job uses explicit cluster and node reservation. Figure 9 shows the results of a multi-cluster reservation for a large block of nodes per cluster at a fixed time of 6:05 AM CET (5:05 UTC), approximately 18 hours after the reservation was made. It was manually confirmed in advance that the requested resources should be available on all clusters, and a reduction in the number of nodes was made to provide

for a margin of unexpectedly failed nodes. The 5 minute offset from the hour was to provide the grid and cluster infrastructures with time to clean and restart the nodes after previous reservations ending at 6 AM were completed. In fact, this result does not include clusters which completely failed to respect the reservation or failed to initiate the job, and also three clusters with abnormal behaviour: two where the leader node failed to initiate the task on the workers, and one where the task commenced three hours late. It can be seen that only five clusters, consisting of approximately half of the 800 cores, were able to start the computation within a minute of the reservation. This poses serious problems for coordinated, prescheduled, pre-planned multi-cluster parallel computations, since it suggests it is difficult to have disparate clusters available during the same time window. The Lille cluster shows grey box for waiting time *after* execution, due to problems with synchronizing data from worker nodes back to leader node and user home directory. Other nodes also have this stage, but it takes $< 1s$ so is not visible at this scale (see *paravent-dev.inisa.fr*).

6 Conclusions and Perspectives

These studies have quantitatively revealed the difficulties with executing coordinated cross-cluster and cross-site parallel computational tasks. The layered state machine model from Section 3.2 for grid jobs has facilitated detailed tracking of state transitions through the various layers of the grid, and been a part of identifying misconfigured clusters and nodes. The metrics defined in Section 4 provide measures which suggest a $90 - 95\%$ occupancy efficiency at the cluster level is reasonable if the clusters are correctly configured and operating normally. Regarding parallel computing at the cluster level, it is clear that heterogeneity is rampant, even when a cluster claims it is composed of identical machines. Latencies in binding grid tasks to particular nodes, and initiation of tasks on a particular core can introduce delays of several seconds to minutes. This presents two major challenges for parallel computing on the grid: i) synchronization of the task start time; and ii) partitioning of the computational load. While static measures of relative performance on a cluster or node level are valuable, it is clear that these cannot always be trusted, hence it is reasonable to imagine the need for dynamic, application-layer determination of node performance prior to the initiation of parallel computations. Ideally this responsibility would be taken by the grid infrastructure (and by implication the cluster owner), however the federated and unreliable nature of grids suggests the user or application needs to manage this for the present. At the level of cross-cluster parallel computing the key challenges are coordinated reservation and start-up. The work here has not yet investigated what granularity of computations are practical, however the start-up delays and unreliable fulfillment of reservations suggest that "contributory best-effort" Monte Carlo simulators may be appropriate, where simulators enter and exit a simulator pool in a dynamic peer-to-peer fashion, and are acquired by a simulation manager and assigned to particular computations "on demand", rather than with a simulation manager expecting a set of simulators based on a prior reservation.

The future work for PicsouGrid is to implement and deploy American option pricing algorithms which factor in the heterogeneous and dynamically varying state of available grid resources, to evaluate the degree of parallelism which can be achieved, and at what cost, and to discover the performance impacts of real cross-cluster communicating parallel computations in a grid environment. It will also be important to evaluate PicsouGrid on other grid infrastructures such as EGEE, who over the last year have put specific effort into the parallel computing capabilities of gLite and the grid middleware[21][22]. Once a foundation for parallel computing on the grid has been established, the operational requirements for on-demand option pricing will need to be evaluated. This will comprise a combination of response time performance of American option pricing algorithms, throughput of a larger multi-user pricing application deployed on the grid, and the dynamicity and fault-tolerance of the application in the presence of changing grid resources.

References

[1] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81(3), pp. 637–654, 1973.

[2] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton, "ProActive: an integrated platform for programming and running applications on grids and P2P systems," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 69–77, 2006.

[3] R. Bolze, F. Cappello, E. Caron, M. Dayd, F. Desprez, E. Jeannot, Y. Jgou, S. Lantri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Ira, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed.," *International Journal of High Performance Computing Applications*, vol. 20, pp. 481–494, Nov. 2006.

[4] L. Baduel, F. Baude, and D. Caromel, "Asynchronous typed object groups for grid programming," *International Journal of Parallel Programming*, 2007.

[5] F. Baude, D. Caromel, A. di Costanzo, C. Delbé, and M. Leyton, "Towards deployment contracts in large scale clusters & desktop grids," in *Invited paper at the Int. Worshop on Large-Scale, volatile Desktop Grids, in conjunction with the IEEE IPDPS conference*, (Denver, Colorado, USA), Apr. 2007.

[6] S. Bezzine, V. Galtier, S. Vialle, F. Baude, M. Bossy, V. Doan, and L. Henrio, "A Fault Tolerant and Multi-

Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing in Finance.," *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[7] CERN, "The LHC Computing Grid Project." `http://lcg.web.cern.ch/LCG/`.

[8] "The Open Science Grid." `http://www.opensciencegrid.org/`.

[9] F. Longstaff and E. Schwartz, "Valuing American options by simulation: a simple least-squares approach," *Review of Financial Studies*, 2001.

[10] Jorge A. Picazo, *Monte Carlo and Quasi-Monte Carlo Methods 2000:: Proceedings of a Conference Held at Hong Kong Baptist University, Hong Kong SAR, China, November 27-December 1, 2000*, ch. American Option Pricing: A Classification-Monte Carlo (CMC) Approach, pp. 422–433. Springer, 2002.

[11] A. Ibanez and F. Zapatero, "Monte Carlo Valuation of American Options through Computation of the Optimal Exercise Frontier," *Journal of Financial and Quantitative Analysis*, vol. 39, no. 2, pp. 239–273, 2004.

[12] I. Toke, "Monte Carlo Valuation of Multidimensional American Options Through Grid Computing," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 3743, p. 462, 2006.

[13] J. Wan, K. Lai, A. Kolkiewicz, and K. Tan, "A parallel quasi-Monte Carlo approach to pricing multidimensional American options," *International Journal of High Performance Computing and Networking*, vol. 4, no. 5, pp. 321–330, 2006.

[14] P. Boyle, A. Kolkiewicz, and K. Tan, "An improved simulation method for pricing high-dimensional American derivatives," *Mathematics and Computers in Simulation*, vol. 62, no. 3-6, pp. 315–322, 2003.

[15] A. Avramidis and Y. Zinchenko and T. Coleman and A. Verma, "Efficiency improvements for pricing American options with a stochastic mesh: Parallel implementation," *Financial Engineering News*, December 2000.

[16] M. Broadie and P. Glasserman, "Pricing American-style securities using simulation," *Journal of Economic Dynamics and Control*, vol. 21, no. 8, pp. 1323–1352, 1997.

[17] W. Stevens *et al.*, *Advanced programming in the UNIX environment*. Addison-Wesley, 1992.

[18] S. Jha and A. Merzky, "GFD-I.71: A Requirements Analysis for a Simple API for Grid Applications," tech. rep., Open Grid Forum, May 2006. Simple API for Grid Applications Research Group.

[19] D. Bell, T. Kojo, P. Goldsack, S. Loughran, D. Milojicic, S. Schaefer, J. Tatemura, , and P. Toft, "Configuration Description, Deployment, and Lifecycle Management," November 2003.

[20] M. Roehrig, W. Ziegler, and P. Wieder, "GFD-I.11: Grid Scheduling Dictionary of Terms and Keywords," tech. rep., Global Grid Forum, November 2002. Grid Scheduling Dictionary Working Group.

[21] Richard de Jong and Matthijs Koot, "Preparing the Worldwide LHC Computing Grid for MPI applications," tech. rep., CERN, June 2006.

[22] "Grid Ireland MPI Technical Computing Group." `http://www.grid.ie/mpi/wiki/`.